Extracting Models of Security-Sensitive Operations using String-Enhanced White-Box Exploration on Binaries

Juan Caballero
Stephen McCamant
Adam Barth
Dawn Song

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-36
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-36.html

March 6, 2009

# Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **06 MAR 2009**	2. REPORT TYPE	3. DATES COVERED **00-00-2009 to 00-00-2009**

4. TITLE AND SUBTITLE **Extracting Models of Security-Sensitive Operations using String-EnhancedWhite-Box Exploration on Binaries**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR’S ACRONYM(S)
	11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Models of security-sensitive code enable reasoning about the security implications of code. In this paper we present an approach for extracting models of security-sensitive operations directly from program binaries, which lets third-party analysts reason about a program when the source code is not available. Our approach is based on string-enhanced white-box exploration, a new technique that improves the effectiveness of current white-box exploration techniques on programs that use strings, by reasoning directly about string operations, rather than about the individual byte-level operations that comprise them. We implement our approach and use it to extract models of the closed-source content sniffing algorithms of two popular browsers: Internet Explorer 7 and Safari 3.1. We use the generated models to automatically find recently studied content-sniffing XSS attacks, and show the benefits of stringenhanced white-box exploration over current byte-level exploration techniques.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**23**	

Extracting Models of Security-Sensitive Operations using String-Enhanced White-Box Exploration on Binaries

Juan Caballero[*][†], Stephen McCamant[†], Adam Barth[†], Dawn Song[†]
[†]UC Berkeley [*]Carnegie Mellon University

Abstract

Models of security-sensitive code enable reasoning about the security implications of code. In this paper we present an approach for extracting models of security-sensitive operations directly from program binaries, which lets third-party analysts reason about a program when the source code is not available. Our approach is based on *string-enhanced white-box exploration*, a new technique that improves the effectiveness of current white-box exploration techniques on programs that use strings, by reasoning directly about string operations, rather than about the individual byte-level operations that comprise them. We implement our approach and use it to extract models of the closed-source content sniffing algorithms of two popular browsers: Internet Explorer 7 and Safari 3.1. We use the generated models to automatically find recently studied content-sniffing XSS attacks, and show the benefits of string-enhanced white-box exploration over current byte-level exploration techniques.

1 Introduction

Extracting a model of security-sensitive code is an important problem because such a model allows an analyst to reason about the security implications of the code. For example, recent work has shown that such models can be used to find subtle XSS attacks, called content-sniffing XSS attacks [9].

Unfortunately, for most programs models of their security-sensitive operations are not available. Thus, it is important to develop techniques that can automatically extract such models. For some programs, a model of their security-sensitive operations can be directly obtained from the source code, but automated techniques to extract the model are still needed when the code base is large. In addition, for a large number of commercial-off-the-self applications (COTS), the source code of the application is not available to an independent analyst and such approach is not possible. Often, when access to the source code is not possible, one still has access to the binary of the application, which is the most accurate representation of the application's functionality because it represents the code that gets executed. In that case, both static and dynamic analysis techniques can be applied to the binary. Finally, there are some cases where neither the source code nor the binary of the application is available to an independent analyst. In that case, if remote access to the application is available (e.g., an XSS filter used by a web server), one can construct the model using black-box testing: observing the outputs generated from selected inputs.

Note that we refer to extracting models of some security-sensitive operations of an application, rather than modeling the complete application. There are two reasons for this. First, for most security analysis we only need to model some security-sensitive part of an application, such as the access control algorithm of an operating system, or the content sniffing algorithm in a browser. Second, applications can range up to many megabytes of binary code. Since model extraction techniques are expensive, it makes sense to focus the analysis only on the relevant parts of the application.

In this paper we present a technique for extracting models of security-sensitive operations in an application. Our technique works directly on the binary of the application and thus can be applied even when the source code of the application is not available. Our technique uses symbolic execution white-box exploration, and is similar in spirit to previous techniques that use symbolic execution for automatic testing [16, 23, 24]. White-box exploration techniques can be used to extract high coverage models that include many paths, compared to previously used single-path models [12]. The advantage of white-box exploration techniques over black-box testing is that the models generated by white-box exploration are of higher quality because black-box testing usually achieves lower coverage, thus relying on heuristics to generalize the model, which may introduce errors.

Many security-sensitive applications heavily rely on string operations, and current exploration approaches are not effective at dealing with such operations. Thus, we propose *string-enhanced white-box exploration*, a technique to enhance the exploration of programs that use strings, by reasoning directly about string operations, rather than reasoning about the individual byte-level operations that comprise those string operations. Our results show that reasoning directly about string operations increases the coverage that the exploration achieves per unit of time.

In this paper, we use our model extraction technique to obtain a model for the closed-source content sniffing algorithms of two different browsers: Internet Explorer 7 and Safari 3.1[1]. Then, we show how the extracted models enable finding filtering-failure attacks.

Filtering-failure attacks. There exists a broad class of security issues where a filter, intended to block malicious inputs destined for an application, incorrectly models how the application interprets those inputs. A filtering-failure attack is an evasion attack where the attacker takes advantage of those differences between the filter's and the application's interpretation of the same input to bypass the filter and still compromise the application.

One class of filtering-failure attacks are content-sniffing XSS attacks [12]. Content sniffing XSS attacks are a class of cross-site scripting (XSS) attacks in which the attacker uploads some malicious content to a benign web site (e.g., a picture uploaded to Wikipedia, or a paper uploaded to a conference management system). The malicious content is accessed by a user of the web site, and is interpreted as HTML by the user's browser. Thus, the attacker can run JavaScript, embedded in the malicious content, in the user's browser in the context of the site that accepted the content. Such attacks are possible because the web site's content filter has a different view than the user's browser about which content should be considered HTML. This discrepancy often occurs due to a lack of information or understanding by the web site's developers about the content sniffing algorithm that runs in the browser and decides what MIME type to associate to some given content. For instance, some content that the web site's filter accepts because it interprets it as a PostScript document might be interpreted as HTML by the browser of the user downloading the content.

There are other examples of filtering-failure attacks. For example, an Intrusion Detection System (IDS) may deploy a vulnerability signature to protect some unpatched application in the internal network. If the signature incorrectly models which inputs exploit the vulnerability in the application, then an attacker can potentially construct an input that is not matched by the IDS' signature but still exploits the application. Also, a network filter may try to block traffic from a peer-to-peer (P2P) application; a failure to correctly model the output messages from the P2P client would allow some traffic to get through.

In this work we build a tool that first extracts the model of the content-sniffing algorithm in the browser, and then uses that model and the model of the web site's content filter to automatically find content-sniffing XSS attacks. For web sites that use the content sniffing functions provided by PHP for content filtering, our tool finds 6 MIME types that an attacker can use to build content-sniffing XSS attacks against Internet Explorer 7, and a different set of 6 MIME types that the attacker can use against Safari 3.1.

[1] Though much of Safari is open-source as part of the WebKit project [7], the content sniffing algorithm in Safari is part of the closed-source *CFNetwork* library.

Contributions. This paper makes the following contributions:

- We propose string-enhanced white-box exploration, a technique for path exploration on program binaries. String-enhanced white-box exploration enables reasoning directly about the string operations performed by the program, rather than the byte-level operations that comprise them, significantly increasing the coverage that the exploration achieves per unit of time on programs that use string operations.
- We design and implement a model extraction tool that employs string-enhanced white-box exploration and use it to extract models of the closed-source content sniffing algorithms for two popular browsers: Internet Explorer 7 and Safari 3.1 We show that the produced models can be used to automatically find filtering-failure attacks, a class of security issues where an attacker exploits the differences between a filter and the program that the filter tries to model.

The rest of the paper is organized as follows. In Section 2 we present the model extraction problem and formally define filtering-failure attacks. In Section 3 we introduce our string-enhanced white-box exploration technique. Then, in Section 4 we describe how the system generates path predicates that contain string constraints. Next, in Section 5 we detail how the system generates new inputs from the path predicates. We evaluate our approach in Section 6. In Section 7 we describe the related work, and finally, we conclude in Section 8.

2 Overview and Problem Definition

In this section, we first give an overview of the model extraction problem. Then we formally define filtering-failure attacks, and finally, we present our running example.

2.1 Model extraction

Extracting a model of a security-sensitive operation of a program is useful for security applications such as finding filtering-failure attacks, where an attacker exploits the fact that a filter designed to protect some application incorrectly models the behavior of that application. In this paper we present a model extraction technique that uses white-box exploration. Our technique works directly on the binary of the application and thus can be generally applied, even when the source code of the application is not available. Model extraction techniques that require access only to the binary program are important because of a large class of commercial-off-the-self applications, for which an independent security analyst has access to the binary of the application but not to the source code.

Our model extraction technique focuses on extracting models of a single security-sensitive operation of an application, rather than a model of the whole application. The reason is that for security analysis we only need to understand the security-sensitive operations, (e.g., the access control algorithm of an operating system or the content sniffing algorithm in a browser), which are usually a small subset of the whole application. Since model extraction techniques can be expensive, it makes sense to focus the analysis only on the relevant parts of the application.

An important characteristic of many security applications, such as an IDS signature matching engine, a content sniffing algorithm in a browser, a host filter to block inappropriate web content, or a spam filtering proxy, is that they rely heavily on string operations. Current white-box exploration techniques [16, 23, 24] are not efficient at dealing with such applications because they contain a large number of loops (potentially unbounded if they depend on the input). Our *string-enhanced white-box exploration* technique improves the exploration of programs that use strings by reasoning directly about string operations, which increases the coverage that the exploration achieves per unit of time.

Using string-enhanced white-box exploration we explore multiple program execution paths, generate a path predicate for each path and then produce a model that is the disjunction of the path predicates. Then we apply the generated models to finding filtering-failure attacks. We define this problem next.

2.2 Problem Definition

Given the filter and a model of the application that the filter tries to model, a filtering-failure attack is an input that is accepted by the filter and can potentially be harmful for the application. Thus, a filtering-failure attack is an evasion attack that bypasses the filter and still can compromise the application. A filter can be modeled as a Boolean predicate ($M_{filter}^{accepted}(x)$) on an input x, which returns true if the input x is considered safe and false if the input is considered dangerous. In our approach the application's processing of the input is also modeled as a Boolean predicate. The exact semantics of the Boolean predicate depend on the application. We explain how to model the application for a content sniffing attack next.

The content sniffing algorithm in the user's browser can be modeled as a deterministic multi-class classifier that takes as input the payload of an HTTP response, the URL of the request, and the response's `Content-Type` header, and produces as output a MIME type for use by the browser. This multi-class classifier can be split into binary classifiers, one per MIME type returned by the content sniffing algorithm, where each binary classifier is a Boolean predicate that returns true if the payload of the HTTP response is considered to belong to that MIME type and false otherwise (for instance $M_{csa}^{text/html}(x)$, or $M_{csa}^{html}(x)$ for brevity). We term such Boolean predicates *submodels* to differentiate them from the complete model of the content sniffing algorithm.

For a content sniffing attack, we seek inputs that are accepted by the web site's content filter and for which the content sniffing algorithm (CSA) of the browser outputs a MIME type that can contain active content, such as `text/html` or `application/x-shockwave-flash`. Thus, we can model the browser's content sniffing algorithm as a binary classifier that returns true if the HTTP payload is considered to belong to any of the dangerous MIME types: $M_{csa}^{html}(x) \vee M_{csa}^{flash}(x)$. To find a content sniffing attack that is accepted by the web site's content filter and interpreted as HTML by the browser, we construct the following query: $M_{filter}^{accepted}(x) \wedge (M_{csa}^{html}(x) \vee M_{csa}^{flash}(x))$. If the solver returns an input that satisfies such query, then we have found a content sniffing attack.

2.3 Running example

Figure 1 shows an example content sniffing algorithm that takes as input the proposed MIME type (*ct*) and the content (*data*), and returns a suggested MIME type. It tries to sniff an HTML document when the proposed MIME type is *text/plain* and JPEG and GIF images if the proposed MIME type is *application/octet-stream*. A possible filtering-failure attack for this algorithm would require a Content-Type of *text/plain* and the content to contain the string $<$*html*$>$, because that is the only option to return a MIME type that can contain active code. An attacker could use the following input to try to bypass a content filter in a website and run JavaScript in the browser:

```
CT: text/plain
DATA: GIF89a<html><script>alert("XSS");</script></html>
```

```
const char* text_mime = "text/plain", binary_mime = "application/octet-stream";
const char* html_mime = "text/html", gif_mime = "image/gif", jpeg_mime = "image/jpeg";

const char* sniff(char *ct, char *data) {
  // Sniff HTML from text/plain
  if (strcmp(ct,text_mime) == 0) {
    if (strstr(data,"<html>") != 0) return html_mime;
    else return text_mime;
  }
  // Sniff GIF, JPEG from application/octet-stream
  if (strcmp(ct,binary_mime) == 0) {
    if ((strncasecmp(data,"GIF87a",6) == 0) || (strncasecmp(data,"GIF89a",6) == 0))
      return gif_mime;
    if ((data[0] == 0xFF) && (data[1] == 0xD8))
      return jpeg_mime;
  }
  return NULL;
}
```

Figure 1: Our running example, a simple content sniffing algorithm that takes as input the proposed MIME type and the raw data, and returns a suggested MIME type.

3 String-Enhanced White-Box Exploration

This section describes how our string-enhanced white-box exploration technique works. First, we present an overview of the white-box exploration process, then we describe the string enhancements introduced by string-enhanced white-box exploration, and finally we present the benefits of our technique.

3.1 White-Box Exploration Overview

Our model extraction technique is based on white-box exploration. Here we briefly describe the overall process and refer the reader to previous work for a more detailed explanation [16, 23, 24].

White-box exploration is an iterative process that incrementally explores new execution paths in the program by generating new inputs that traverse those paths. In each iteration, also called a *round* or a *test*, an input is sent to the program under analysis, running inside a *symbolic execution module*, which performs a mixture of symbolic and concrete execution. The symbolic execution module runs the program on the given input and outputs a *path predicate*, a Boolean predicate on the input that captures how the input was processed in that particular execution. In addition, the symbolic execution module also outputs other information about the execution such as the test result, e.g., the MIME type returned by the *sniff* function in our running example, the coverage of the execution, or any errors detected during the execution.

The path predicate is a conjunction of constraints on the input that captures how the input was processed along that execution path. Given the path predicate, the *input generator* produces a new input by negating one constraint in the path predicate and asking a solver to produce an input that satisfies the new predicate with the negated constraint. This process can be repeated for each constraint in the path predicate. Since many inputs can be generated from each path predicate, and many path predicates will be generated during the exploration, the *path selector* is in charge of assigning priorities to the newly generated inputs and selecting the input with the highest priority to start a new round of the iterative process.

The whole iterative process is started with an initial input, called the *seed*, and runs iteratively until there are no more paths to explore, or a user-specified maximum run-time is reached.

3.2 String Enhancements

Overall, our string processing comprises four steps. First, the symbolic execution module replaces constraints generated inside string functions with constraints on the output of those string functions. Then, the constraints on the output of the string functions are translated into abstract string operators. Next, the system represents each string as an array of some maximum length and a length variable, and translates the abstract string operators into a representation that is understood by an off-the-self solver that supports theory of arrays and integers. Finally, the system uses the answer of the solver to build an input that starts a new iteration of the exploration.

Our string handling has been designed to abstract the underlying representation of the strings, so that it can be used with programs written in different languages. So far, we have applied our approach to both C strings, where the string is represented as a null-terminated array of characters, and C++ string libraries, where the string may be represented as an object containing an array of characters and an explicit length.

One important characteristic of C/C++ strings is that one can operate with them using string functions such as *strlen* or *strcmp*, but also directly access the underlying array of characters. Thus, the path predicate generated by the symbolic execution module may contain both constraints on the output of string functions and constraints on the individual bytes that comprise the string. We explain how the path predicate is generated in Section 4 and the abstract string syntax in Section 5.

3.3 Benefits of Adding String Support to the Exploration

When extracting a model we would like to focus our efforts on some part of the program which contains the unknown security-sensitive functionality, usually some function and all code used or called from that function. In addition, we would like to limit the model to the functionality that is truly unknown, excluding parts of the code that are commonly used and may be well understood and commonly used functions in publicly available shared libraries.

String functions are a sweet spot in that 1) they appear in many programs, 2) some types of programs use them heavily (e.g., parsers or filters), 3) they are important for a large number of security sensitive programs such as host or network filters, 4) they contain loops, which in the worst case may be unbounded if they depend on some external input, 5) their prototype is usually known, or can be obtained with limited work, and 6) they are easy to reason about (as compared for example to system calls where one might have to reason about the underlying operating system or even the hardware).

Being able to reason directly about the output of string functions increases the coverage that the exploration achieves per unit of time. The increase in coverage is due to eliminating the time spent exploring inside the string functions, and reducing the size of the queries to the solver.

4 Generating the Path Predicate

In this section, we present how the symbolic execution module produces a path predicate that contains string constraints.

The symbolic execution module runs the program on both concrete and symbolic inputs and outputs the path predicate. In addition to the symbolic execution itself, the symbolic execution module has to perform two additional tasks: introducing the string symbols when the function to be explored is called, and creating new symbols for the output of predefined string functions. Next, we introduce the concept of function hooks and describe how they are used by the symbolic execution module to accomplish both tasks.

Function hooks. The symbolic execution module lets the user specify code stubs that are executed when a certain function is called by the monitored program (before any instruction in the function is executed) [2]. We term the pair of the function name, and the code stub that gets executed when the function is invoked, a *function hook*. A function hook can also define a *return hook*, another code stub that is executed when the function returns (after the return instruction in the function is executed). Function hooks can be used to perform different actions such as making the inputs of a function symbolic or summarizing a function by turning off symbolic execution in the function and creating new symbols for the output of the function. Function hooks require access to the function's prototype, so that the code stub can locate the parameters of the function in the stack and the return values of the function if a return hook is defined.

Introducing string symbols. To start the symbolic execution, the system sets a function hook for the function to be explored (i.e., the *sniff* function in our running example). When the function is called, the code stub performs the following operations: 1) reads the parameters of the function from the stack, 2) determines the length of the input strings defined by the user (i.e., the *ct* and *data* parameters of the *sniff* function in our running example), 3) adds to the symbolic context the memory locations comprising each input string, and 4) sets a return hook. When the function returns, the return hook logs the return values of the function (i.e., the suggested MIME type) and stops the symbolic execution.

When creating a new string symbol, the representation of the string is abstracted. In particular, the function hook uses the function's prototype to determine whether the input strings are null-terminated arrays of characters or objects containing an array and an explicit length variable. If the string is null-terminated the location of the null-character does not become symbolic. If the string is an object with an explicit length variable then, in addition to the memory locations that comprise the string, the length variable also becomes a symbol.

Introducing string constraints. To introduce string constraints the system uses function hooks for some predefined string functions. The function hooks for the string functions differ from the functionality described above. To distinguish between both types of function hooks, we term the function hooks for string functions, *function summaries*. A function summary performs the following operations: 1) reads the parameters of the function from the stack, 2) checks if any of the parameters of the function is symbolic; if none are symbolic then it returns, 3) turns off symbolic execution inside the function, so that no constraints will be generated inside a summarized function, 4) sets up a return hook. When the string function returns, the return hook makes the return values of the function symbolic.

Currently, the symbolic execution module provides function summaries for over 100 string functions for which prototypes are publicly available. The prototypes of those functions can be found, among others, at the Microsoft Developer Network [4], the WebKit documentation [7], or the standard C library [6].

The user is expected to provide a function summary for any function that is currently not available in the framework. When a program to be explored uses functions that have no publicly available prototype, some manual reverse engineering of the binary is needed to extract the function's prototype. For example, the content sniffing algorithm in Internet Explorer 7 uses two string functions that have no publicly available prototype: *shlwapi.dll::Ordinal_151* and *shlwapi.dll::Ordinal_153*. Our analysis found that *shlwapi.dll::Ordinal_151* is a case sensitive comparison of some maximum length, which can use the existing function summary for *msvcrt.dll::strncmp*. For *shlwapi.dll::Ordinal_153* we found that it is a case sensitive version of the *shlwapi.dll::Ordinal_151* and again we could reuse an existing function summary.

[2] The framework supports specifying the functions by name, to avoid having to update the function's start address when the module that contains the function is loaded at an address that is not the default one. The system also supports specifying the function by ordinal if it is not exported by name.

The time spent doing such analysis was close to an hour per function. Once obtained, the function summaries are added to the framework so that they can be reused in the future.

String function classes. The symbolic execution module groups string function summaries into classes, where two string functions in the same class would generate the same path predicate if the specific function called in the source code was replaced with any other function in the same class. For example, *msvcrt.dll::strstr* and *msvcr71.dll::strstr* both belong to the same *STRSTR* string function class. Grouping string function summaries into classes reduces the number of types of constraints that the system needs to translate into a format understood by the solver. Currently, the framework supports 14 classes of string functions: *STRSTR*, *STRCMP*, *STRCASECMP*, *STRNCMP*, *STRLEN*, *STRNCASECMP*, *COMPARESTRING*, *CFEQUAL*, *STRCPY*, *STRNCPY*, *STRCHR*, *MEMCHR*, *WCTOMB*, and *MBTOWC*.

To summarize, during symbolic execution, every time one of the predefined string functions is called, the symbolic execution module introduces new symbols for the output of the string function. Then, when the program uses those symbols (e.g., in a comparison) a string constraint is introduced in the path predicate. The path predicate output by the symbolic execution module contains a mixture of string constraints (i.e., on the output of the string functions), and constraints on some of the bytes of the input strings, which can be generated by either functions that are not summarized or direct access to the bytes belonging to the string, as shown in line 18 of our running example.

5 Solving Constraints

In this section we describe how our system generates from one path predicate a number of inputs that can be used to start new iterations of the exploration.

At the beginning of this project, no publicly available solver supported strings as first-class types. Thus, to solve the string constraints in the path predicate the system needs to be able to translate them into a notation that is understood by our off-the-self solver, STP [21]. To this purpose, we leverage the fact that many solvers, including STP, have support for array and integer theories, and represent each string (i.e., input strings plus any strings derived from them, for example through *strcpy*) as a pair of an array of some given maximum size and a length variable[3].

The translation comprises two steps. First, the system translates the string constraints to an intermediate syntax, which abstracts the representation of the strings and defines a common set of functions and predicates that operate on strings. We call this representation the *abstract string syntax*. Then, the system represents each string as a pair of an array of some maximum size and a length variable, and translates the operators in the abstract string syntax to constraints on the corresponding arrays and length variables.

Although no string constraint solvers were available, we designed our abstract string syntax so that we could easily make use of such solvers whenever they become available. Simultaneous work reports on solvers that support a theory of strings [10, 26, 29]. Given our design, rather than translating the abstract string operations into a theory of arrays and integers, we could as well generate constraints in a theory of strings instead, benefiting from the performance improvements provided by these specialized solvers.

[3] The system does not fundamentally depend on the "Modulo Theories" part of STP being an SMT solver. Thus, at the cost of some syntactic convenience (and integration with non-string constraints) a pure SAT solver could be used.

Functions		
(strlen String)	$S \to I$	(strlen s) returns the length of s
(substr String Int Int)	$S \times I \times I \to S$	(substr s i j) returns the substring of s starting at position i and ending at position j (inclusive)
(strcat String String)	$S \times S \to S$	(strcat s_1 s_2) returns the concatenation of s_1 and s_2
(strupper String)	$S \to S$	(strupper s) returns an uppercase version of s
(strncopy String Int)	$S \times I \to S$	(strncopy s i) returns a string of length i which equals the first i characters of s
(strfromwide String)	$S \to S$	(strfromwide s) returns a narrow character version of s
(strtostring Char)	$C \to S$	(strtostring c) returns a string containing only the character c
(chrat String Int)	$S \times I \to C$	(chrat s i) returns the character at position i in string s
(chrupper Char Char)	$C \to C$	(chrupper c) returns an uppercase version of c
Predicates		
(strcontains String String)	$S \times S \to B$	(strcontains s_1 s_2) returns true if s_2 is a substring of s_1 at any position
(strcontainsat String String Int)	$S \times S \times I \to B$	(strcontainsat s_1 s_2) returns true if s_2 is contained in s_1 starting at position i in s_1
(= String String)	$S \times S \to B$	(= s_1 s_2) returns true if s_1 is equal to s_2
(distinct String String)	$S \times S \to B$	(distinct s_1 s_2) returns true if s_1 is not equal to s_2
(strlt String String)	$S \times S \to B$	(strlt s_1 s_2) returns true if s_1 is lexicographically less-than s_2
(strle String String)	$S \times S \to B$	(strle s_1 s_2) returns true if s_1 is lexicographically less-or-equal s_2
(strgt String String)	$S \times S \to B$	(strgt s_1 s_2) returns true if s_1 is lexicographically greater-than s_2
(strge String String)	$S \times S \to B$	(strge s_1 s_2) returns true if s_1 is lexicographically greater-or-equal s_2
(strcaseequal String String)	$S \times S \to B$	(strcaseequal s_1 s_2) returns true if s_1 is equal to s_2 case-insensitive
(strcasedistinct String String)	$S \times S \to B$	(strcasedistinct s_1 s_2) returns true if s_1 is not equal to s_2 case-insensitive
(strcaselt String String)	$S \times S \to B$	(strcaselt s_1 s_2) returns true if s_1 is lexicographically less-than s_2 case-insensitive
(strcasele String String)	$S \times S \to B$	(strcasele s_1 s_2) returns true if s_1 is lexicographically less-or-equal s_2 case-insensitive
(strcasegt String String)	$S \times S \to B$	(strcasegt s_1 s_2) returns true if s_1 is lexicographically greater-than s_2 case-insensitive
(strcasege String String)	$S \times S \to B$	(strcasege s_1 s_2) returns true if s_1 is lexicographically greater-or-equal s_2 case-insensitive

Table 1: Abstract string syntax.

5.1 Abstract String Syntax and Translations

In this section we first present our abstract string syntax, then we describe how the user defines the input strings, and finally we present the translations of the string constraints to the abstract string syntax, and from the abstract string syntax to the notation understood by our solver.

Abstract string syntax. The abstract string syntax represents the minimal interface we would like a solver, using strings as first-order types, to provide. Table 1 presents the description of the functions and predicates that comprise our abstract string syntax. The strings in the abstract string syntax are immutable. Thus, string operations such as modifying a string, copying a string, translating the string to upper case or concatenating two strings, always return a new string.

In our abstract string syntax each string can be seen as a variable-length array, where each element of the array has no encoding and is of fixed length[4]. Having no string encoding enables support for both binary and text strings. For simplicity, we term each element of the array a *character*, even if they may represent binary data. For text strings, an element of the array can be seen as a Unicode code-point[5].

Case-insensitive operators rely on the *chrupper* function, which forms the basis for *strupper*. Our current *chrupper* function uses the ASCII uppercase conversion (i.e., only code points U+0061 ('a') through U+007a ('z') have an uppercase version). We plan to enhance this function to represent the Unicode *uppercase* character property. Note that it is considered a valid operation to apply the case-insensitive functions to binary strings, as programs may (either incorrectly or abusing the semantics of the function) perform such operations.

[4] Our implementation uses 16-bit integers to represent a character. Although a 16-bit integer is not enough to hold all Unicode code points, it is enough for the applications we consider. Each character could be represented as a 32-bit integer if all Unicode code points are needed.

[5] A Unicode code-point is different from a *grapheme*, which is closer to what end-users consider as characters. For example a character with a dieresis (e.g., ä) is a grapheme, but could be encoded as two Unicode code points.

String constraint	**Abstract String Syntax**
STRCMP(s_1,s_2) = 0 ; COMPARESTRING(s_1,s_2) = 2; CFEQUAL(s_1,s_2) = 1; s_2 = STRCPY(s_1)	= s_1 s_2
STRCMP(s_1,s_2) $\neq$ 0 ; COMPARESTRING(s_1,s_2) $\neq$ 2; CFEQUAL(s_1,s_2) = 0	distinct s_1 s_2
STRCMP(s_1,s_2) $<$ 0; COMPARESTRING(s_1,s_2) $<$ 2	strlt s_1 s_2
STRCMP(s_1,s_2) $>$ 0; COMPARESTRING(s_1,s_2) $>$ 2	strgt s_1 s_2
STRSTR(s_1,s_2) $\neq$ 0	strcontains s_1 s_2
STRSTR(s_1,s_2) $=$ 0	not (strcontains s_1 s_2)
STRCASECMP(s_1,s_2) = 0	strcaseequal s_1 s_2
STRCASECMP(s_1,s_2) $<$ 0	strcaselt s_1 s_2
STRCASECMP(s_1,s_2) $>$ 0	strcasegt s_1 s_2
STRNCMP(s_1,s_2,n) = 0	= (substr s_1 0 $(n-1)$) (substr s_2 0 $(n-1)$)
STRNCMP(s_1,s_2,n) $<$ 0	strlt (substr s_1 0 $(n-1)$) (substr s_2 0 $(n-1)$)
STRNCMP(s_1,s_2,n) $>$ 0	strgt (substr s_1 0 $(n-1)$) (substr s_2 0 $(n-1)$)
STRNCASECMP(s_1,s_2,n) = 0	strcaseequal (substr s_1 0 $(n-1)$) (substr s_2 0 $(n-1)$)
STRNCASECMP(s_1,s_2,n) $<$ 0	strcaselt (substr s_1 0 $(n-1)$) (substr s_2 0 $(n-1)$)
STRNCASECMP(s_1,s_2,n) $>$ 0	strcasegt (substr s_1 0 $(n-1)$) (substr s_2 0 $(n-1)$)
STRCHR(s,c) $\neq$ 0	strcontains s (strtostring c)
STRCHR(s,c) $=$ 0	not (strcontains s (strtostring c))
MEMCHR(s,c,n) $\neq$ 0	strcontains (substr s 0 $(n-1)$) (strtostring c)
MEMCHR(s,c,n) $=$ 0	not (strcontains (substr s 0 $(n-1)$) (strtostring c))
s_2 = STRNCPY(s_1,n)	= s_2 (substr s_1 0 $(n-1)$)
s_2 = MBTOWC(s_1)	= s_2 s_1
s_2 = WCTOMB(s_1)	= s_2 (strfromwide s_1)

Table 2: Translation of string constraints to the abstract string syntax.

Predicate	**Translation**
= s_1 s_2	$l(s_1) = l(s_2) \wedge \bigwedge_{i=0}^{i=l(s_1)-1} s_1[i] = s_2[i]$
strcaseequal s_1 s_2	$l(s_1) = l(s_2) \wedge \bigwedge_{i=0}^{i=l(s_1)-1} chrupper(s_1[i]) = chrupper(s_2[i])$
strcontains s_1 s_2	$\bigvee_{i=0}^{i=l(s_1)-1}(l(s_1) \geq l(s_2) + i) \wedge (\bigwedge_{j=0}^{j=l(s_2)-1} s_1[i+j] = s_2[j])$
= s_2 substr s_1 i j	$l(s_2) = j - i + 1 \wedge \bigwedge_{k=0}^{k=j-i} s_2[k] = s_1[i+k]$
= s strtostring c	$l(s) = 1 \wedge s[0] = c$
= s_2 (strfromwide s_1)	$l(s_2) = l(s_1) \wedge \bigwedge_{i=0}^{i=l(s_1)}(s_1[i] < 256 \wedge s_2[i] = s_1[i])$

Table 3: Predicate translation. For simplicity, the negation of the above predicates is not shown.

All encoding is removed when converting to the abstract string syntax. For example, conversions from UTF-8 to UTF-16 and viceversa, used by the content sniffing algorithm in Internet Explorer 7 for the *Content-Type* string, are handled during the translation to the abstract string syntax. Note that, while widening conversions (e.g., UTF-8 to UTF-16) are straightforward to handle, narrowing conversions (e.g., UTF-16 to UTF-8) can be lossy, and thus need a special conversion function (*strfromwide*). Our current implementation for *strfromwide* only handles conversions when all characters in the string belong to the ASCII charset, which is enough for programs that take as input ASCII strings.

Input strings. For each program to explore, the user provides some information about the input strings, namely the name to assign to the string symbol, the maximum size of the string, and the type of the string.

The maximum size of each input string needs to be conservatively chosen, otherwise the solver might not be able to solve some constraints. For example, in our running example the user provides the name and the maximum size for the *ct* and *data* strings. If the user selected a maximum length of 16 bytes for the *ct* string then the constraint generated in line 11 would be unsolvable since *ct* could not equal *application/octet-stream*. On the other hand the tighter the maximum length, the less time that it will take the solver to find a satisfying answer, if there is one. For some programs, such as content sniffing algorithms, the maximum length of the input strings is known. For example, previous work has shown that the maximum length of the content sniffing buffer, which corresponds to the *data* string in our running example, is 1024 bytes for Safari 3.1, and 256 bytes for Internet Explorer 7 [9]. In our experiments we use those values for the *data* string and set the maximum length of the the Content-Type (*ct*) string to be 64 bytes.

Our current implementation supports three types of input strings: *ASCII strings*, *UTF-16 strings*, and *binary strings*. For example, in our running example the *ct* string is an ASCII string, while the *data* string is a binary string. These classes are just offered as a convenience to the user, and can be used to alert the user if the program uses uncommon string operations, such as case-insensitive operations on strings that the user defines as binary strings.

Translating to the abstract string syntax. Table 2 presents the translation from the constraints generated on the output of the 14 classes of supported string functions to the abstract string syntax. Table 2 shows one of the benefits of using an abstract string syntax: constraints from functions with different prototypes but similar functionality (e.g, STRCMP(s_1,s_2) $<$ 0, COMPARESTRING(s_1,s_2) $<$ 2), can be translated to the same basic string operation (e.g., lexicographical less-than).

Constraints on individual bytes are translated using the character extraction operator, *chrat*[6]. For example, the constraint `if (data[0] == ’x’)` $\{\ldots\}$, would be translated as `(chrat data 0) = 0x78`. This is possible because the symbolic execution module knows for each memory location if it belongs to a symbolic string and the offset into the string, which can be used to identify the character index.

In our running example, if the function *sniff* is run with the following inputs:

```
CT: application/octet-stream
DATA: GIF89a\000\000
```

the path predicate translated to the abstract string syntax would be:

```
(distinct ct "text/plain") &&
(= ct "application/octet-stream") &&
(strcasedistinct (substr data 0 5) "GIF87a") &&
(strcaseequal (substr data 0 5) "GIF89a")
```

where the first constraint corresponds to the false branch in the conditional on line 6 of the running example, the second to the true branch of the conditional on line 11, and the final two correspond to the two clauses in the conditional on line 12 (false and true branches respectively). The value returned by the function is *image/gif*.

Translating from the abstract string syntax. Table 3 shows how the constraints on the output of the summarized string functions are translated to the theory of arrays and integers. Each string variable s is represented by its length $l(s)$, its maximum length $ml(s)$, and an array of bytes $s[i]$, where the index i ranges from 0 to $ml(s) - 1$. The maximum lengths $ml(s)$ are translation-time constants, but the lengths $l(s)$ may not be, so unless the length of a string is constant, bounds that are shown involving $l(s)$ are in fact translated by expanding them up to a bound based on $ml(s)$ and guarding with additional conditions on $l(s)$. Note that the translation shown for strfromwide is restricted to the case of 8-bit code-points; a more complex translation would be needed for applications that involved other characters.

For example, the first constraint in the above path would be translated as:

$\neg((ct\_len = 10) \wedge (ct[0] = `t`) \wedge (ct[1] = `e`) \wedge (ct[2] = `x`) \wedge (ct[3] = `t`) \wedge (ct[4] = `/`) \wedge (ct[5] = `p`) \wedge (ct[6] = `l`) \wedge (ct[7] = `a`) \wedge (ct[8] = `i`) \wedge (ct[9] = `n`))$

[6] Currently, we do not deal with unaligned accesses such as reading a single byte from a UTF-16 string, but such accesses could be translated as extracting the character corresponding to the offset being accessed and then masking the other byte.

```
HTTP/1.1·200·OK\n
Server:·Apache/2.2.4·(Fedora)\n
Content-Type:·text/plain\n\n
<html>·This· is·html·</html>\n\n
```

Figure 2: A complete input with the input strings highlighted.

where ct_len is the length integer that represents the length of the ct array. Note that the solver only understands about integers, but we use the text representation of the character here for the reader's benefit (e.g., the constraint would use 0x74 instead of ‘t‘). The last constraint in the above path would be translated as:

$(ct\_len \geq 6) \wedge (chrupper(ct[0]) = \text{‘}G\text{‘}) \wedge (chrupper(ct[1]) = \text{‘}I\text{‘}) \wedge (chrupper(ct[2]) = \text{‘}F\text{‘}) \wedge (chrupper(ct[3]) = \text{‘}8\text{‘}) \wedge (chrupper(ct[4]) = \text{‘}9\text{‘}) \wedge (chrupper(ct[5]) = \text{‘}a\text{‘})$

Additional constraints. The symbolic execution module adds some additional constraints to each query to the solver. For each input string defined by the user, it adds a constraint to force the length of the string to be between zero and the predefined maximum length of the string, $0 \leq l(s) \leq ml(s)$. In addition, for ASCII strings it adds constraints to force each byte in the string to belong to the ASCII charset, $\bigwedge_{i=0}^{i=ml(s)-1} 0 \leq s[i] \leq 127$. A special case happens when converting to the abstract string syntax a string constraint from a function that assumes the input strings to be null-terminated, such as the string functions in the C library. In this case the symbolic execution module adds some additional constraints to the path predicate to exclude the null character from the possible code values. This prevents the solver from producing inputs that actually violate the generated length constraints. If our running example had the following constraint: `if (strstr(ct, "html") == 0)` { ... }, then if the null character is allowed to be part of the *ct* string, then the solver could return the following satisfying assignment for *ct*: $l(ct) = 6 \wedge s[0] = \text{‘}a\text{‘} \wedge s[1] = \text{‘}\backslash 0\text{‘} \wedge s[2] = \text{‘}h\text{‘} \wedge s[3] = \text{‘}t\text{‘} \wedge s[4] = \text{‘}m\text{‘} \wedge s[5] = \text{‘}l\text{‘}$. Given the null terminated representation expected by *strstr*, that string would have an effective length of 1 character, and the generated input would not traverse the true branch of the conditional.

5.2 Input Generation

Once the solver returns a satisfying assignment for a query, the system needs to generate a new input that can be sent to the program, so that another round of the exploration can happen.

However, the values from the symbolic strings might not completely define an input. For example, Figure 2 shows a complete input used in the exploration of the content sniffing algorithm of Safari 3.1, where the inputs strings are highlighted and the spaces have been replaced by dots. In this case, the system uses a parser to break the seed message into fields, and lets the user specify which fields correspond to the input strings. Once the solver returns a satisfying assignment for the MIME type and the content, the system generates a new input with the same field structure as the seed message. In the new input, the input strings have been replaced with the values returned by the solver. All other fields in the new input reuse the values from the seed message. Our default parser is the Wireshark protocol parser. For other inputs that are not supported by this parser, the system lets the user provide an input file with the format of the seed input.

Generating an input that reaches the entry point. The function being explored might be run in the middle of some longer execution. To guarantee that the generated inputs will reach the function under study, we need to add all constraints on the input strings generated by the code that executes before the function under study, as additional constraints to each query to the solver. For example, when analyzing the content sniffing algorithm in Safari, we need to add any constraints on the Content-Type header or the HTTP payload that occur in the execution before the content sniffing function is called. To identify such constraints we run the symbolic execution module making the whole HTTP message symbolic[7]. All constraints on the input strings before the call to the content sniffing function are included as additional constraints. Such constraints may include, among others, parsing constraints that require the MIME type string not to contain any HTTP delimiters such as end of line characters, or constraints that force the Content-Type value to be one of a list of MIME types that trigger the content sniffing algorithm.

6 Evaluation

In this section we present our evaluation results. First we introduce our setup, then we show statistics from the extracted models, next we compare string-enhanced white-box exploration with previous byte-level white-box exploration, and finally we describe some examples of filtering-failure attacks that we find using the extracted models.

6.1 Setup

We have extracted models from the content sniffing algorithm of two major browsers, for which source code is not available: Safari 3.1 and Internet Explorer 7. In both cases we have evaluated the browser running on a Windows XP Service Pack 3 operating system.

In addition, we have manually written a model for the signatures used by the Unix *file* tool [1]. The Unix file tool is an open-source command line tool, deployed in many Unix systems, which given a file outputs its MIME type and some associated information. The signatures of the Unix file tool are used by the MIME detection functions in PHP. Those functions in turn are used by the content filter of many web sites. For example, the MIME detection functions from PHP are used by popular open-source code such as Mediawiki [5], which is used by Wikipedia to handle uploaded content.

As described in Section 4, a prerequisite for the exploration is to identify the prototype of the function that implements the content sniffing algorithm, as well as the string functions used by that function, which are not already supported by the symbolic execution module. To this end we use available documentation, commercial off-the-self tools [8], as well as our own binary analysis tools [35]. We describe this step next.

Content sniffing is performed in Internet Explorer 7 by the function *FindMimeFromData* available in the *urlmon.dll* library [3]. We obtain the function prototype, including the parameters and return values, from the Microsoft Developer Network (MSDN) documentation [2]. We had to write function summaries for two string functions used by FindMimeFromData for which our framework did not have support and for which the prototype is not publicly available: *shlwapi.dll::Ordinal_151* and *shlwapi.dll::Ordinal_153*[8].

Although a large portion of Safari 3.1 is open-source as part of the WebKit project, the content sniffing algorithm is implemented in *CFNetwork.dll*, the networking library in the Mac OS X platform, which is not part of the WebKit project. In addition to extracting the prototype of the content sniffing function, we also had to add to the symbolic execution module two function summaries for functions that have a publicly available prototype: *CoreFoundation.dll::CFEqual* and *CoreFoundation.dll::CFStringCompare*.

[7] Since there may exist multiple paths to the content sniffing algorithm, we might have to rerun this step with different inputs. One indication to rerun this step is if during the exploration the tool reports that some inputs are not reaching the content sniffing algorithm (i.e., empty path predicates).

[8] *shlwapi.dll* is the Shell Light Weight Utility Library, a Windows library that contains functions for URL paths, registry entries, and color settings.

Model	Seeds	Path count	% HTML paths	Avg. # Paths per seed	Avg. Time per path	# Inputs generated	Avg. path depth	# blocks found	Avg. # blocks per seed
Safari 3.1	7	1558	12.4%	222.6	16.8 sec	7166	12.1	205	193.9
Internet Explorer 7	7	948	8.6%	135.4	26.6 sec	64721	212.1	450	388.5

Table 4: Model statistics.

Since the *CoreFoundation.dll* library provides the fundamental data types, including strings, which underlie the MacOS X framework, these function summaries can be reused by many other applications that use this framework.

6.2 Model Extraction

In this section we present some statistics about the models automatically extracted from the content sniffing algorithms of Internet Explorer 7 and Safari 3.1. We term the process of exploring from one seed until no more paths are left to explore or a user-specified maximum run-time is reached, an *exploration run*.

Each model is created by combining multiple exploration runs, each starting from a different seed. To obtain the seeds we first select some common MIME types and then we randomly choose one file of each of those MIME types from the hard-drive of one of our workstations. For our experiments each exploration run lasts 6 hours and the seeds come from 7 different MIME types: application/java, image/gif, image/jpeg, text/html, text/vcard, video/avi, video/mpeg. The same seeds are used for both browsers.

Table 4 summarizes the extracted models. The table shows the number of seeds used in the exploration, the number of path predicates that comprise each model, the percentage of path predicates in the previous column where the content sniffing algorithm returned the MIME type *text/html*, the average number of paths per seed, the average time in seconds needed to generate a path predicate, the number of inputs generated, the average number of branches in each path (i.e., the path depth), the number of distinct program blocks discovered during the complete exploration from the 7 seeds, and the average number of blocks discovered per seed.

The number of paths that return *text/html* is important because those paths form the *text/html* submodel, which we use in Section 6.4 to find filtering-failure attacks. Two other MIME types that can be considered dangerous are *application/pdf* and *application/x-msdownload*. Both are only sniffed by the content sniffing algorithm in Internet Explorer 7. For simplicity, we focus on content-sniffing XSS attacks, where the attacker embeds JavaScript in some content that the content sniffing algorithm interprets as HTML.

The content sniffing algorithm in Safari 3.1 is smaller because it has signatures for 10 MIME types, while the content sniffing algorithm in Internet Explorer 7 contains signatures for 32 different MIME types. This is shown in Table 4 by shorter path predicates that require less time to be produced. The longer path predicates for Internet Explorer 7 also explain why the number of inputs generated for Internet Explorer 7 is almost an order of magnitude larger than for Safari 3.1.

Exploring from multiple seeds helps increase the coverage for Internet Explorer 7 because the content sniffing algorithm in Internet Explorer 7 decides which signatures to apply to the content depending on whether it considers the content to be text or binary data. Thus, it is more efficient to do one exploration run for 6 hours starting from a binary seed (e.g., application/pdf) and another exploration run for 6 hours from a text seed (e.g., text/html) than to do a single exploration run for 12 hours starting from either a binary or a text seed. We have not observed this effect in Safari 3.1. We discuss more about how to compute the number of blocks discovered in the next section.

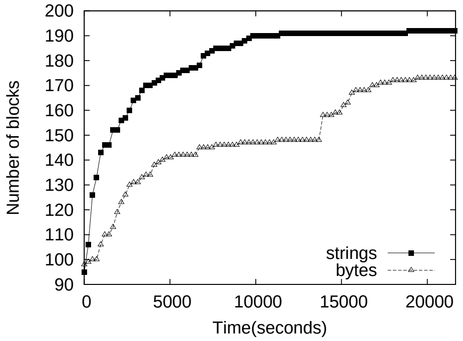

Figure 3: String-enhanced white-box exploration versus byte-level white-box exploration on the Safari 3.1 content sniffing algorithm. Each curve represents the average number of blocks discovered for 7 exploration runs each starting from a different seed and running for 6 hours.

6.3 Coverage

In this section we demonstrate the benefits of string-enhanced white-box exploration, when compared to byte-level white-box exploration. First, we detail how we measure the number of blocks discovered and then present the coverage results.

Methodology. During the execution, every time an unconditional jump, conditional jump, call, or return instruction is seen, the symbolic execution module stores the address of the next instruction to be executed, which represents the first instruction in a block (i.e., the *block address*). The number of distinct block addresses is our coverage metric. This approach may underestimate the number of blocks discovered[9], but gives a reasonable approximation of the blocks covered by the exploration without requiring static analysis of the binary to extract all basic blocks. A difference with fuzzing approaches is that we do not want to maximize coverage of the whole program, only of the function that implements the content sniffing algorithm. Thus, we are not interested in measuring coverage in auxiliary functions such as memory allocation functions (e.g., malloc), string functions (e.g., strcmp), or synchronization functions on critical sections. Our goal is to only count blocks inside functions for which no prototype is publicly available. Our tool approximates this behavior by automatically ignoring blocks inside functions that appear in the list of functions exported by name.

Coverage results. Figure 3 shows the number of blocks that the system discovers over time on the Safari 3.1 content sniffing algorithm, when the exploration uses strings (square line) and when we disable the string processing and the path predicate only contains byte-level constraints (triangle line). Each curve represents the average number of blocks discovered for 7 exploration runs each starting from a different seed and lasting 6 hours. The *strings* curve corresponds to the 7 exploration runs from which the model of the content sniffing algorithm in Safari 3.1 was extracted (shown in Table 4), while the *bytes* curve is the average of 7 byte-level exploration runs starting from the same seeds used for extracting the model.

The graph shows that the string-enhanced white-box exploration achieves higher coverage than the byte-level exploration on the same amount of time. Thus, it better employs the resources associated to the exploration. This happens because 61.6% of all byte-level constraints occur inside string functions. Thus,

[9] When compared to counting basic blocks in a control-flow graph, our approach may underestimate the number of basic blocks because one block found during execution could be represented as multiple basic blocks in the control-flow graph. This happens when some path contains a jump whose target location is in the middle of one block previously discovered dynamically. In the CFG this case counts as two basic blocks while dynamically, since we deal with each path separately, it only counts as one.

the byte-level exploration expends considerable time exploring inside the string functions, and no new blocks in the content sniffing algorithm are discovered during that time.

The statistics from the byte-level exploration show that the average path depth for the byte-level exploration is 4.8 times larger than the average path depth for the string-level exploration (as shown in Table 4). Thus, the average size of the queries to the solver is also larger for the byte-level exploration. The statistics also show that the number of queries to the solver is 79% higher for the byte-level exploration. Thus, using string-enhanced white-box exploration reduces the number of queries to the solver and the average size of those queries. This reduces the computation done by the solver, where a significant part of the exploration time is spent, which is why previous work has spent significant time in optimizations to reduce it [16,23,24].

6.4 Filtering-Failure Attacks

Filtering-failure attacks require finding an input (e.g., an image) that is accepted by the web filter and interpreted by the content sniffing algorithm in the browser as a privileged MIME type such as *text/html*. For this experiment we use the manually created model for the Unix file tool [1], which corresponds to the content filter of websites that use the standard MIME detection functions in PHP, and the models extracted from the content sniffing algorithms of both Internet Explorer 7 and Safari 3.1, presented in Section 6.2.

For each MIME type supported by the Unix *file* tool and at least one of the browsers, the tool queries the solver for an input that satisfies the *file* signature but is interpreted as *text/html* by the content sniffing algorithm in the browser.

Internet Explorer. We find content-sniffing XSS attacks for 6 different MIME types: *application/postscript*, *audio/x-aiff*, *image/gif*, *image/tiff*, *text/xml*, and *video/mpeg*. Here, we show two examples of such documents. Querying the solver for an input that is accepted as *application/postscript* by the Unix file tool and as *text/html* by the content sniffing algorithm in Internet Explorer 7 returns the following answer:

```
CT: application/pdf
DATA: %!t?HPTw\nOtKoCglD<HeadswssssRsD
```

The first 2 bytes of the DATA string, *“%!”*, satisfy the *application/postscript* signature in the Unix file tool, and they also satisfy the *application/postscript* signature in the content sniffing algorithm of Internet Explorer 7. In addition, the input returned by the solver contains the substring *<Head*, which satisfies the *text/html* signature used by the content sniffing algorithm in Internet Explorer 7. But, because the content sniffing algorithm in Internet Explorer 7 tests the *text/html* signature before the *application/postscript* signature, then the content would be interpreted as *text/html* by Internet Explorer 7.

Querying the solver for an input that is accepted as *audio/x-aiff* by the Unix file tool and as *text/html* by the content sniffing algorithm in Internet Explorer 7 returns the following answer:

```
CT: image/gif
DATA: <htmLpflAIFF\t\t\t\227\t\t\t\003\t\008\201\t
```

The first 5 bytes of the DATA string, *<htmL*, satisfy one of the HTML signatures used by Internet Explorer 7 for *text/html*, while bytes 8 – 11, *AIFF*, satisfy the *audio/x-aiff* signature for the Unix file tool. This input would not match Internet Explorer’s *audio/x-aiff* signature which is:

```
(strncmp(DATA,"MROF",4) == 0) ||
((strncmp(DATA,"FORM",4) == 0) &&
 ((strncmp(DATA[8],"AIFF",4) == 0) || (strncmp(DATA[8],"AIFC",4) == 0)))
```

(the Content-Type is not included in the signature because the content sniffing algorithm in Internet Explorer 7 applies all signatures if the Content-Type matches any of 37 “supported” MIME types). Since the input does not match the signature, it will be interpreted as *text/html* by Internet Explorer 7.

Safari. We find content-sniffing XSS attacks against Safari 3.1 for 6 different MIME types: *application/postscript*, *audio/x-aiff*, *image/gif*, *image/png*, *image/tiff*, and *video/mpeg*. One of the examples found is the following chameleon MPEG document:

```
CT: <EMPTY>
DATA: \000\000\001\187MmM\129\000\002\002TLT\001L\002\001\000<hTMl>e\000\000
```

Here, the solver returns an empty Content-Type, and some content where the first four bytes satisfy the *video/mpeg* signature of the Unix file tool, and the tag <*hTMl*> satisfies the *text/html* signature used by the content sniffing algorithm in Safari.

7 Related Work

In this section we first present previous work on automatic testing and automatic signature generation, which is related to our work in that they also use white-box exploration or related symbolic execution techniques. Then, we introduce previous work that verifies security properties using software model checking techniques, which requires models of the programs to be verified, and can benefit from automated techniques to extract such models. Next, we describe previous research on cross-site scripting attacks, including content-sniffing XSS attacks, which we automatically find in this work. Finally, we introduce simultaneous work on solvers that support a theory of strings, which can benefit our work.

Automatic testing. Previous work on automatic testing is another application of white-box exploration [15, 16, 23, 24]. There are two main differences between our model extraction technique using string-enhanced white-box exploration and previous work on automatic testing. First, the goal is different: the goal of automatic testing is to find bugs in a program, while the goal of our model extraction is to generate an accurate representation of a program that can be used for reasoning about its security implications. Second, the white-box exploration techniques used by previous work on automatic testing are not efficient on programs that heavily rely on string operations, which are the main target of our string-enhanced white-box exploration. Xu et al. [36] augment white-box exploration with length abstraction, which allows a tool to reason about the length of a string independent of its contents; as in our approach, this is enabled by function summaries. Length abstraction is productive in searching for buffer overflows, but would not be useful for the string-heavy applications we consider, in which the complete contents of strings are critical. Concurrently, our research group is also investigating enhancements to symbolic execution to automatically capture the behavior of loops [33]. These techniques could be used to replace the manually written string function summaries we use here, among other applications, but integrating them with a string decision procedure as in this report is future work. Previous work has also proposed improvements to white-box exploration techniques to reduce the number of paths that need to be explored (i.e., the path explosion problem) by using a compositional approach [22] or trying to identify parts of paths that have already been explored [11]. Such techniques can be combined with our string-enhanced white-box exploration technique to further enhance the exploration.

Automatic signature generation. Previous work on automatic signature generation produces symbolic-execution based vulnerability signatures directly from the vulnerable program binary [13, 14, 18, 19]. Such signatures model the conditions on the input required to exploit a vulnerability in the program. The difference between those signatures and our models is that vulnerability signatures try to cover only paths to a specific program point (namely, the vulnerability) rather than all paths inside some given function. A significant shortcoming of early proposals is that the signatures have low coverage, typically covering a single execution path [19]. More recent approaches have proposed to cover more execution paths by removing unnecessary conditions using path slicing techniques [18], iteratively exploring alternate paths to the vulnerability and adding them to the signature [13, 18], or using static analysis techniques [14].

Property verification. Model checking techniques can be used to determine whether a formal model (including of a program) satisfies a property [17]. They have been applied to security problems such as verifying temporal-logic properties of an access control system [27], and evaluating attack scenarios in a network that contains vulnerable applications [32, 34]. But such techniques typically require the availability of a model, which limits the applicability to other security problems. In this paper we present a technique to automatically extract models from binaries, which can enable the application of model checking techniques to other applications.

Cross-site scripting attacks. Cross-site scripting (XSS) attacks, where an attacker injects active code (e.g., JavaScript) into HTML documents, are an important and widely studied class of attacks [20, 25, 28, 30, 31]. Recent work has studied content sniffing XSS attacks, a class of XSS attacks where the attacker embeds executable code into different types of content, such as images or PDF documents [9]. In this work we show how to construct content-sniffing XSS attacks by automatically extracting models of the content sniffing algorithm in the browser and comparing them to the web site's content filter.

String constraint solvers. Simultaneous work reports on solvers that support a theory of strings [10, 26, 29]. Even though during the course of this work no string constraint solver was publicly available, we designed our abstract string syntax so that it could use such a solver whenever available. Thus, rather than translating the abstract string operations into a theory of arrays and integers, we could easily generate constraints in a theory of strings instead, benefiting from the performance improvements provided by these specialized solvers.

8 Conclusion

In this paper we present an approach for extracting models of security-sensitive operations directly from program binaries. Such models enable third-party analysts to reason about the code, even when the source code of the program is not available. For example, they can be used for finding filtering-failure attacks, a broad class of security issues where a filter, intended to block malicious inputs destined for an application, incorrectly models how the application interprets those inputs, leaving the application vulnerable to attack.

Since many security-sensitive applications rely heavily on string operations, and current exploration approaches are not effective at dealing with such operations, our model extraction approach is based on *string-enhanced white-box exploration*, a novel technique that improves the effectiveness of white-box exploration on programs that use strings by reasoning directly about the string operations. Our results show that reasoning directly about string operations significantly increases the coverage of the exploration per unit of time.

String-enhanced white-box exploration introduces string constraints into the path predicate and translates those constraints into a syntax that can be understood by an off-the-self solver that supports a theory of arrays and integers. It abstracts the strings representation so it can be used, for instance, with programs that use C strings (where the string is represented as a null-terminated array of characters), as well as C++ string libraries (where the string may be represented as an object containing an array of characters and an explicit length).

We implement our model extraction approach using string-enhanced white-box exploration and use it to extract models of the proprietary content sniffing algorithms of two popular browsers: Internet Explorer 7 and Safari 3.1. We use the generated models to automatically find content-sniffing XSS attacks, an instance of filtering-failure attacks. We also show how string-enhanced white-box exploration increases the effectiveness of the exploration compared to byte-level exploration techniques.

9 Acknowledgements

We would like to thank Rhishikesh Limaye, Susmit Jha, and Sanjit A. Seshia who collaborated in the design of the abstract string syntax and with whom we had many helpful discussions.

This material is based upon work partially supported by the National Science Foundation under Grants No. 0311808, No. 0448452, No. 0627511, and CCF-0424422, and by the Air Force Office of Scientific Research under MURI Grant No. 22178970-4170. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Air Force Office of Scientific Research, or the National Science Foundation.

References

[1] Fine Free File Command. `http://darwinsys.com/file/.`

[2] MSDN: FindMimeFromData Function. `http://msdn.microsoft.com/en-us/library/ms775107(VS.85).aspx.`

[3] MSDN: MIME Type Detection in Internet Explorer. `http://msdn.microsoft.com/en-us/library/ms775147.aspx.`

[4] Microsoft Developer Network. `http://msdn.microsoft.com.`

[5] Sites Using MediaWiki/en. `http://www.mediawiki.org/wiki/Sites_using_MediaWiki/en.`

[6] The ISO/IEC 9899:1999 C Programming Language Standard. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf.`

[7] The WebKit Open Source Project. `http://webkit.org.`

[8] The IDA Pro Disassembler and Debugger. `http://www.hex-rays.com/idapro/.`

[9] A. Barth, J. Caballero, and D. Song. Secure Content Sniffing for Web Browsers *or* How to Stop Papers from Reviewing Themselves. *IEEE Symposium on Security and Privacy,* Oakland, California, May 2009.

[10] N. Bjorner, N. Tillmann, and A. Voronkov. Path Feasibility Analysis for String-Manipulating Programs. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems,* York, United Kingdom, March 2009.

[11] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking Path Explosion in Constraint-Based Test Generation. *Conference on Tools and Algorithms for the Construction and Analysis of Systems,* Budapest, Hungary, March 2008.

[12] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation. *USENIX Security Symposium,* Boston, Massachusetts, August 2007.

[13] D. Brumley, J. Newsome, D. Song, H. Wang, and Somesh Jha. Towards Automatic Generation of Vulnerability-Based Signatures. *IEEE Symposium on Security and Privacy,* Oakland, California, May 2006.

[14] D. Brumley, H. Wang, S. Jha, and D. Song. Creating Vulnerability Signatures Using Weakest Pre-Conditions. *IEEE Computer Security Foundations Symposium,* Venice, Italy, July 2007.

[15] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *Symposium on Operating System Design and Implementation,* San Diego, California, November 2008.

[16] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: A System for Automatically Generating Inputs of Death Using Symbolic Execution. *ACM Conference on Computer and Communications Security,* Alexandria, Virginia, October 2006.

[17] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, 2000.

[18] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing Software By Blocking Bad Input. *Symposium on Operating Systems Principles,* Bretton Woods, New Hampshire, October 2007.

[19] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. *Symposium on Operating Systems Principles,* Brighton, United Kingdom, October 2005.

[20] M. Cova, V. Felmetsger, D. Balzarotti, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. *IEEE Symposium on Security and Privacy,* Oakland, California, May 2008.

[21] V. Ganesh and D. Dill. A Decision Procedure for Bit-Vectors and Arrays. *Computer Aided Verification Conference,* Berlin, Germany, August 2007.

[22] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-Based Whitebox Fuzzing. *SIGPLAN Conference on Programming Language Design and Implementation,* Tucson, Arizona, June 2008.

[23] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. *SIGPLAN Conference on Programming Language Design and Implementation,* Chicago, Illinois, June 2005.

[24] P. Godefroid, M. Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. *Network and Distributed System Security Symposium,* San Diego, California, February 2008.

[25] M. V. Gundy and H. Chen. Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks. *Network and Distributed System Security Symposium,* San Diego, California, February 2009.

[26] P. Hooimeijer and W. Weimer. A Decision Procedure for Subset Constraints Over Regular Languages. *SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, June 2009.

[27] S. Jha, N. Li, M. V. Tripunitara, Q. Wang, and W. H. Winsborough. Towards Formal Verification of Role-Based Access Control Policies. *IEEE Transactions on Dependable and Secure Computing,* 5(4), Oct.-Dec. 2008.

[28] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. *International World Wide Web Conference,* Banff, Canada, May 2007.

[29] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A Solver for String Constraints. MIT CSAIL, Technical Report MIT-CSAIL-TR-2009-004, February 2009.

[30] M. Martin and M. S. Lam. Automatic Generation of XSS and SQL Injection Attacks with Goal-Directed Model Checking. *USENIX Security Symposium,* San Jose, California, July 2008.

[31] Y. Nadji, P. Saxena, and D. Song. Document Structure Integrity : A Robust Basis for XSS Defense. *Network and Distributed System Security Symposium,* San Diego, California, February 2009.

[32] R. W. Ritchey and P. Ammann. Using Model Checking to Analyze Network Vulnerabilities. *IEEE Symposium on Security and Privacy,* Oakland, California, May 2001.

[33] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-Extended Symbolic Execution on Binary Programs. UC Berkeley, Technical Report UCB/EECS-2009-34, March 2009.

[34] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated Generation and Analysis of Attack Graphs. *IEEE Symposium on Security and Privacy,* Oakland, California, May 2002.

[35] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A New Approach to Computer Security Via Binary Analysis. *International Conference on Information Systems Security,* Hyderabad, India, Keynote invited paper, December 2008.

[36] R.-G. Xu, P. Godefroid, and R. Majumdar. Testing for Buffer Overflows with Length Abstraction. *International Symposium on Software Testing and Analysis,* Seattle, Washington, July 2008.